NLP second task

project - https://github.com/LiorBreitman8234/genreDetectionProject

Lior Breitman – 212733257
Talia Seada – 211551601
Netanel Levine – 312512619

Part A: Results of training over softmax without hidden layers
After training the softmax logistic regression for 20000 iterations, we got an accuracy rating off

1. training set: 65.9%
2. validation set: 64.2%
3. Test set: 63.6%

These results are far better than expected Because when looking at the models that the FMA git had, we saw that most of the results for accuracy were between 35 and 50 percent.
When using the gradient descent optimizer, we got an accuracy of around 30%, with the loss being over 100 at each iteration
After switching to the Adam optimizer, our results got way better, with the loss dropping to around 5 and accuracy going up by more than 30%.
We played with the alpha hyperparameter when we took bigger one the results where smaller and unsetifying so we took smaller one while insuring we dont get overfitting, eventually we chose to take alpha of size - 0.0001.

code:

```
features = 518
categories = 161  # 161
x = tf.placeholder(tf.float32, [None, features])
y_ = tf.placeholder(tf.float32, [None, categories])
W = tf.Variable(tf.zeros([features, categories]))
b = tf.Variable(tf.zeros([categories]))

z = tf.matmul(x, W) + b
pred = tf.nn.softmax(z)
# loss = -tf.reduce_mean(y_ * tf.log(pred))
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(y_, z))

update = tf.train.AdamOptimizer(0.0001).minimize(loss)
```

```
data_x = data
targets = OneHotEncoder().fit_transform(inx).toarray()
```

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
```

```
def calc_acc_train():
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    acc = sess.run(accuracy, feed_dict={x: X_train, y_: y_train})
    print("Accuracy:", acc * 100)
    return acc

def calc_acc_valid():
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    acc = sess.run(accuracy, feed_dict={x: X_valid, y_: y_valid})
    print("Validation Accuracy: ", acc * 100, "%")
    return acc

def calc_acc_test():
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    acc = sess.run(accuracy, feed_dict={x: X_test, y_: y_test})
    print("Test Accuracy:", acc * 100, "%")
    return acc
```

```
max_acc = 0
itr = 0
for i in range(0, 20001):
    sess.run(update, feed_dict={x: X_train, y_: y_train})
    if i % 1000 == 0:
        print('Iteration:', i, '\n',
              ' W:', sess.run(W), '\n',
              ' b:', sess.run(b), '\n',
              'loss:', loss.eval(session=sess, feed_dict={x: X_train, y_: y_train}))
        acc = calc_acc_train()
        if max_acc < acc:
            max_acc = acc
            saver.save(sess, "mySoftmax", global_step=1)
            itr = i
```

```
print(max_acc)
print(itr)
```

```
0.659131
20000
```

```
valid_acc = calc_acc_valid()
```

```
Validation Accuracy:  64.21371102333069 %
```

```
test_acc = calc_acc_test()
```

```
Test Accuracy: 63.62903118133545 %
```

Part B: Results of training over MLP (with hidden layers).

option 1: 3 hidden layers

After training the MLP with 4 hidden layers for 10000 iterations, we got an accuracy rating off

4. training set: 61.0%
5. validation set: 58.6%
6. Test set: 58.4%

When using the gradient descent optimizer, we got an accuracy of around 20%, with huge loss, so we used the adam optimizer and as we can see it improved the results.

We played with the alpha hyperparameter, at first we used 0.001 and it got us overfitting (with the layers in sizes 129, 129, 129), we had 95% in the training set but 53% in the validation set, so when we changed it to 0.01 it gave us the wanted results.

We also played with the layers sizes,we tries different sizes to each layer it gave us overfitting and we tried more different sizes, but it gave us lower results.

Eventually we chose these sizes -129, 129, 129, and this alpha - 0.01.

code:

```
#Defining the initial weights and input/output

(first_layer_size, second_layer_size, third_layer_size) = [518 // 4, 518 // 4, 518 // 4]
x = tf.placeholder(tf.float32, [None, 518])
y = tf.placeholder(tf.float32, [None, 161])

weight_first = tf.Variable(tf.truncated_normal([518, first_layer_size], stddev=0.3, dtype=tf.float32))
bias_first = tf.Variable(tf.constant(0, shape=[first_layer_size], dtype=tf.float32))
activation_first = tf.nn.relu6(tf.matmul(x, weight_first) + bias_first)

weight_second = tf.Variable(tf.truncated_normal([first_layer_size, second_layer_size], stddev=0.3, dtype=tf.float32))
bias_second = tf.Variable(tf.constant(0, shape=[second_layer_size], dtype=tf.float32))
activation_second = tf.nn.relu6(tf.matmul(activation_first, weight_second) + bias_second)

weight_third = tf.Variable(tf.truncated_normal([second_layer_size, third_layer_size], stddev=0.3, dtype=tf.float32))
bias_third = tf.Variable(tf.constant(0, shape=[third_layer_size], dtype=tf.float32))
activation_third = tf.nn.relu6(tf.matmul(activation_second, weight_third) + bias_third)

weights_output = tf.Variable(tf.truncated_normal([third_layer_size, 161], stddev=0.3, dtype=tf.float32))
bias_output = tf.Variable(tf.constant(0, shape=[161], dtype=tf.float32))
```

```
#creating prediction, training and cross-entropy methods
pred = tf.nn.softmax(tf.matmul(activation_third, weights_output) + bias_output)
loss = -tf.reduce_sum(y * tf.log(tf.clip_by_value(pred, 1e-10, 1.0)))
training_step = tf.train.AdamOptimizer(0.01).minimize(loss)
```

```
#setting session and starting to run
training_sess = tf.Session()
training_sess.run(tf.global_variables_initializer())
```

```
#training loop:
saver = tf.train.Saver()
epochs = 10001
batch_size = 389
iterations = int((x_train.shape[0]) / batch_size)
maxAccuracy = 0
for epoch in range(epochs):
    ptr = 0
    for iteration in range(iterations):
        batch_input = x_train[ptr:ptr + batch_size]
        batch_label = y_train[ptr:ptr + batch_size]
        ptr = (ptr + batch_size) % x_train.shape[0]
        _, err = training_sess.run([training_step, loss], feed_dict={x: batch_input, y: batch_label})
    if epoch % 1000 == 0:
        currAccuracy = training_sess.run(accuracy, feed_dict={x: x_train, y: y_train})
        valid_accuracy = training_sess.run(accuracy, feed_dict={x: x_valid, y: y_valid})
        if currAccuracy > maxAccuracy:
            saver.save(training_sess, "best_mlp_3_layers", global_step=1)
            maxAccuracy = currAccuracy
        print(f"epoch: {epoch} - accuracy for epoch {currAccuracy*100} - max accuracy: {maxAccuracy*100}")
        print(f"validation set accuracy: {valid_accuracy*100}")
```

```
train_accuracy = training_sess.run(accuracy, feed_dict={x: x_train, y: y_train})
print(train_accuracy)
```

```
0.61026263
```

```
valid_accuracy = training_sess.run(accuracy, feed_dict={x: x_valid, y: y_valid})
print(valid_accuracy)
```

```
0.58689517
```

```
test_accuracy = training_sess.run(accuracy, feed_dict={x: x_test, y: y_test})
print(test_accuracy)
```

```
0.5840726
```

option 2: 4 hidden layers

After training the MLP with 4 hidden layers for 10000 iterations, we got an accuracy rating off

7. training set: 61.1%
8. validation set: 59.0%
9. Test set: 59.7%

When using the gradient descent optimizer, we got an accuracy of around 20%, with huge loss, so we used the adam optimizer and as we can see it improved the results.
We played with the alpha hyperparameter, at first we used 0.001 and it got us overfitting (with the layers in sizes 259, 129, 64, 32), we had 99% in the training set but 58% in the validation set, so when we changed it to 0.01 it gave us the wanted results.
We also played with the layers sizes, we tried all equal sizes it gave us the same results, we tried small layers sizes first and more different sizes, but it gave us lower results.
Eventually we chose these sizes - 259, 129, 64, 32, and this alpha - 0.01.

code:

```
#Defining the initial weights and input/output

(first_layer_size, second_layer_size, third_layer_size, forth_layer_size) = [518 // 2, 518 // 4, 518 // 8, 518 // 16]
x = tf.placeholder(tf.float32, [None, 518])
y = tf.placeholder(tf.float32, [None, 161])

weight_first = tf.Variable(tf.truncated_normal([518, first_layer_size], stddev=0.3, dtype=tf.float32))
bias_first = tf.Variable(tf.constant(0, shape=[first_layer_size], dtype=tf.float32))
activation_first = tf.nn.relu6(tf.matmul(x, weight_first) + bias_first)

weight_second = tf.Variable(tf.truncated_normal([first_layer_size, second_layer_size], stddev=0.3, dtype=tf.float32))
bias_second = tf.Variable(tf.constant(0, shape=[second_layer_size], dtype=tf.float32))
activation_second = tf.nn.relu6(tf.matmul(activation_first, weight_second) + bias_second)

weight_third = tf.Variable(tf.truncated_normal([second_layer_size, third_layer_size], stddev=0.3, dtype=tf.float32))
bias_third = tf.Variable(tf.constant(0, shape=[third_layer_size], dtype=tf.float32))
activation_third = tf.nn.relu6(tf.matmul(activation_second, weight_third) + bias_third)

weight_forth = tf.Variable(tf.truncated_normal([third_layer_size, forth_layer_size], stddev=0.3, dtype=tf.float32))
bias_forth = tf.Variable(tf.constant(0, shape=[forth_layer_size], dtype=tf.float32))
activation_forth = tf.nn.relu6(tf.matmul(activation_third, weight_forth) + bias_forth)

weights_output = tf.Variable(tf.truncated_normal([forth_layer_size, 161], stddev=0.3, dtype=tf.float32))
bias_output = tf.Variable(tf.constant(0, shape=[161], dtype=tf.float32))
```

```
#creating prediction, training and cross-entropy methods
pred = tf.nn.softmax(tf.matmul(activation_forth, weights_output) + bias_output)
loss = -tf.reduce_sum(y * tf.log(tf.clip_by_value(pred, 1e-10, 1.0)))
training_step = tf.train.AdamOptimizer(0.01).minimize(loss)
```

```
#setting function for correct prediction and accuracy
correct = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

```
#training loop:
saver = tf.train.Saver()
epochs = 10001
batch_size = 389
iterations = int((x_train.shape[0]) / batch_size)
maxAccuracy = 0
for epoch in range(epochs):
    ptr = 0
    for iteration in range(iterations):
        batch_input = x_train[ptr:ptr + batch_size]
        batch_label = y_train[ptr:ptr + batch_size]
        ptr = (ptr + batch_size) % x_train.shape[0]
        _, err = training_sess.run([training_step, loss], feed_dict={x: batch_input, y: batch_label})
    if epoch % 1000 == 0:
        currAccuracy = training_sess.run(accuracy, feed_dict={x: x_train, y: y_train})
        valid_accuracy = training_sess.run(accuracy, feed_dict={x: x_valid, y: y_valid})
        if currAccuracy > maxAccuracy:
            saver.save(training_sess, "best_mlp_4_layers", global_step=1)
            maxAccuracy = currAccuracy
        print(f"epoch: {epoch} - accuracy for epoch {currAccuracy*100} - max accuracy: {maxAccuracy*100}")
        print(f"validation set accuracy: {valid_accuracy*100}")
```

```
train_accuracy = training_sess.run(accuracy, feed_dict={x: x_train, y: y_train})
print(train_accuracy)
```

```
0.61129594
```

```
valid_accuracy = training_sess.run(accuracy, feed_dict={x: x_valid, y: y_valid})
print(valid_accuracy)
```

```
0.5907258
```

```
test_accuracy = training_sess.run(accuracy, feed_dict={x: x_test, y: y_test})
print(test_accuracy)
```

```
0.5975807
```

In conclusion, the model that gave us the best results was the softmax without hidden layers with 63.6%.